(1)

Competitive Management of Distributed Shared Memory

David L. Black
Carnegie-Mellon University
Pittsburgh, PA 15213

Anoop Gupta and Wolf-Dietrich Weber
Stanford University
Stanford, CA 94305

Abstract

This paper presents and analyzes algorithms for managing the distributed shared memory present in non-uniform memory access multiprocessors and related systems. The competitive properties of these algorithms guarantee that their performance is within a small constant factor of optimal even though they make no use of any information about memory reference patterns. Both hardware and software implementation concerns are covered. A case study of the Mach operating system indicates that integration of these algorithms into operating systems does not pose major problems. On the other hand, hardware support is required to obtain the full functionality of the algorithms. We also sketch possible algorithm extensions to additional hardware architectures and software programming models. Reprints. (JES)

Trace driven simulations are used to evaluate our approach and compare it to other alternatives. Speedups of 5 to 10 over random assignment of pages on production applications are achieved without modifying the applications for *non-uniform memory access* (NUMA) architectures. We compare our proposed hardware support with the more aggressive approach of *fully-consistent caches*. An additional factor of 2 to 3 in performance can be obtained from the cache approach, but at the cost of much more hardware. These results indicate that the algorithms and their hardware support may represent a viable cost/performance tradeoff.

1 Introduction

The widespread use of *uniform memory access* (UMA) multiprocessors has sparked interest in using uniform shared memory programming models on *non-uniform memory access* (NUMA) multiprocessors. Use of a common programming model enhances the portability of applications among such machines, and can reduce the effort required to fit or tune applications to NUMA multiprocessors. New techniques are required to manage the distributed physical memory found in a NUMA multiprocessor because the location of memory used by an application (with respect to the processor(s) executing the application) directly affects performance. Optimizing the use of physical memory to minimize access costs is a major issue that must be faced by any implementation of a shared memory programming model on such machines. This paper presents techniques and algorithms for this problem, along with preliminary performance results from trace-driven simulations.

Our algorithms are competitive in a strict theoretic sense. An informal statement of this property is that the algorithms are essentially the best that can be achieved in the absence of information about future memory reference behavior. The techniques of competitive algorithm analysis are particularly useful for this work because they explicitly address the constant factors ignored by standard complexity analysis, and because they are well-suited to the analysis of resource management problems. Previous work has developed competitive algorithms for the related problems of optimizing the use of snoopy caches [8].

The performance results for these algorithms are based on trace-driven simulations of several production applications from UMA multiprocessors. These results show that the proposed algorithms attain total speedups of 5 to 10 over random assignment of pages. This indicates that significant locality (both code and data) may exist in a large class of multiprocessor applications, and that this locality can be detected and exploited automatically. As a result such applications may not require extensive design changes or modifications for use on NUMA multiprocessors; no such changes or modifications were made to our applications.

This paper concentrates on the application aspects of our work. Proofs of the competitive properties of the algorithms can be found in [2]. The next section presents a basic model that covers the systems to which our algorithms are applicable. This is followed by an introduction to competitive algorithms. Section 4 breaks down the basic problem and presents our competitive algorithms for solving it. Sections 5 and 6 continue with a discussion of implementation concerns including the difficulties imposed by most current hardware. Section 7 presents our performance results from trace driven simulations. Sections 8, 9, and 10 briefly discuss extensions of this work. Sections 11 and 12 conclude the paper with a review of related work and a short summary of results.

2 Basic Model

This section presents the basic memory model for which our algorithms were developed. We assume an idealized machine composed of processor-memory clusters, with physical memory divided entirely among the clusters. A processor-memory cluster consists of one or more processors with local memory that is equally accessible (in terms of latency) to all processors. Our idealized machine has two distinct memory access latencies; the latency to access memory in the same cluster, and a significantly larger latency to access memory in another cluster. As a result all memory within a single cluster is equivalent, and all processors within a cluster have identical memory access characteristics (latency in terms of the cluster in which the accessed memory is located). Finally all memory locations outside the cluster have the same access latency from any processor in the cluster.

This basic model subdivides memory into pages and pages into locations. Pages are the fundamental unit of memory management; locations are the fundamental unit of memory access. We assume the existence of virtual memory map-

To appear in COMPCON '89

ping mechanisms, and therefore distinguish between virtual pages (in the address space of some program or the operating system) and physical pages (actual memory in the clusters). Mapping virtual pages to physical pages is one of the responsibilities of a memory management facility. Sharing may result in more than one virtual page in one or more address spaces being mapped to the same physical page. The page size used by our algorithms can be no smaller than the hardware page size if mapping is used, but it may be a multiple thereof.

We normalize our model by assuming the difference in cost between an in-cluster memory access and a remote-cluster memory access is 1; this cost includes the effects of both increased latency and use of interconnection bandwidth. This cost only applies to accesses that actually use the interconnection network; if caches are present at the processors, we only consider accesses that miss in or bypass the appropriate cache. In addition, we are assuming that read and write costs are identical; all of our work generalizes to cases in which these costs are not identical.

This model permits us to analyze techniques for managing the performance impact of distribution in a shared memory system. We concentrate on two major tools for this management: replication and migration of virtual memory. Replication consists of making a copy of a virtual page in another cluster and updating mappings that benefit from this copy (in reduced access time). Migration consists of moving a virtual page from one cluster to another and updating all mappings to that page. We formalize the costs of replication and migration as r and m respectively in terms of access costs. These costs include latency and overhead components, but do not include the additional costs of allocating a physical page in a cluster with a page shortage (i.e. causing pageout) or the additional benefits of freeing a physical page in such a cluster (i.e. avoiding pageout). We separate the issues involved in page reclaim from migration and replication; these are addressed in section 5.1.

Our basic model applies to any machine that can implement NUMA memory. This includes NUMA machines that implement the model directly (e.g. Butterfly [5]), *no remote memory access* (NORMA) machines with uniform access costs, and network shared memory implementations on networks with uniform communication costs. For the last two classes of the machines, it is essential that the system (hardware and/or software) support access forwarding so that accesses to pages that are not in local memory can be satisfied at remote memory *without* moving the entire page to local memory (an expensive operation). Most current NORMA machines (e.g. hypercubes) and network shared memory implementations [4,9,21] do not support this functionality.

3 Basic Problem

The problem we address here is the management of distributed shared memory in architectures conforming to our model. For architectures utilizing a single copy of the operating system (NUMA multiprocessors), this includes not only memory shared explicitly, but also memory shared implicitly via copy-on-write techniques. Since we rely on replication and migration to perform this management, the problem can be restated as "When and under what circumstance should (virtual) pages be replicated into or migrated to memory in other clusters?"

There is a significant difference between this problem and the related problem of snoopy caching: our model and its realizations do not have broadcast, invalidate, or snooping mechanisms that can maintain consistency among multiple copies of a virtual page when writes occur. This prohibits replication of writable pages. Because we have separated the issue of page reclamation, migration of read-only pages make little sense; replication is less costly, and provides the benefits of local access to two clusters instead of one. As a result the overall problem splits into two sub-problems:

- Replication of read-only pages.
- Migration of writable pages.

If a virtual page is both read-only and writable at different times during the execution of an application, we consider each segment (read-only or read/write) of the page's existence to be a separate instance of the corresponding problem.

4 Basic Algorithms

Effective use of replication and migration presents an enigma. Replicating or migrating a page that will never be referenced again is very costly, but so is failing to replicate or migrate a page that will be used heavily in a remote cluster. Avoiding these situations seems to require knowledge of the future that is not available when decisions must be made; this results in a situation where any decision about replication or migration could be both wrong and costly. Problems that require these decisions to be made (affected by future system behavior, but must be made without any knowledge about this behavior) and algorithms that make these decisions are called *on-line*.

Results obtained from the analysis of competitive algorithms provide a solution to this enigma. An on-line algorithm is called *competitive* if its cumulative cost on any sequence is within a constant factor of the cost of the optimal algorithm[1] on the same sequence, and no such algorithm exists for any smaller constant. Competitive algorithms have been found for a number of problems, including list management [16], snoopy caching [8], and some server problems [10]. This paper extends past work by presenting competitive algorithms for replication and migration of distributed shared memory.

4.1 Replication

The on-line replication problem consists of determining when in a sequence of accesses a page should be replicated into other clusters, without look-ahead. Under our model all clusters are uniformly equidistant; if a page is not resident locally, the cost to access it does not depend on the cluster in which it is accessed. As a result the decision to replicate a page into a given cluster is independent of the decisions to replicate into any other clusters. Hence the general replication problem reduces to the replication problem for two clusters with the page initially resident in only one cluster. Algorithm R is our algorithm for this problem.

Algorithm R:

> Count remote accesses from the cluster that does not have the page. When this count exceeds the replication cost, r, replicate the page into the cluster.

[1]The optimal algorithm may look at the entire sequence before making any decisions

Results:

1. Any on-line algorithm for this problem must have a cost that is at least twice the cost incurred by an optimal off-line algorithm on some sequence of accesses.
2. Algorithm **R** is competitive. i.e. its cost is always within a factor of two of optimal on any sequence of accesses.

Algorithm **R** (and algorithm **M** to be presented later) are algorithms that perform well across the entire spectrum of possible sequences. If the specific sequence that will occur is known in advance, an on-line algorithm can be constructed that performs well on that particular sequence, but will perform worse than our algorithm on many other sequences. This embodies the optimality property of our competitive algorithms; they are essentially the best possible in the absence of knowledge about what will happen in the future.

4.2 Migration

The on-line migration problem consists of determining when in a sequence of accesses a page should be migrated to another cluster without look-ahead. Unlike the replication problem, migration depends on the number of clusters: of all the clusters that would benefit from having the page, only one can actually have the page. Decisions to migrate different pages are still independent, so the migration problem reduces to migration of a single page in response to accesses to that page. Algorithm **M** is our algorithm for this multiple cluster page migration problem.

Algorithm M:

> Associate a counter with each cluster: initialize the counts to zero. Access from a cluster that does not have the page increments that cluster's counter, and decrements some other cluster's counter, but not to less than zero. When a cluster's counter reaches twice the migration cost (i.e. $2m$) migrate the page to that cluster and zero its counter. Access from a cluster that has the page decrements some other cluster's counter, but not to less than zero.

All of the counters for a page will be zero after a migration due to the way they are maintained by algorithm **M**.

Results:

1. Any on-line algorithm for this problem must have a cost that is at least three times the cost incurred by an optimal off-line algorithm on some sequence of accesses.
2. Algorithm **M** is competitive, i.e. its cost is always within a factor of three of optimal on any sequence of accesses.

5 Operating Systems Issues

There are two sets of operating systems issues that must be addressed in implementing our algorithms: (i) how do we take into account the limited size of physical memory; and (ii) what are the interactions between the proposed algorithms and the memory management portion of an operating system. The second issue arises primarily if the algorithms are implemented in the operating system kernel; this is an attractive choice both because it permits direct access to mapping information and also because it makes the resulting benefits available to all applications on the system, instead of just those that are modified to use our algorithms.

5.1 Limited Physical Memory Size

Since there are many other demands on physical memory besides those generated by replication and migration (e.g. memory allocation, file mapping, internal use by the operating system, etc.), extending the replication and migration algorithms to control memory usage is not appropriate. We believe that the operating system should separate reuse of physical memory (pageout or page reclaim) from replication and migration issues. Even the fallback position of dedicating a fixed amount of physical memory to replication/migration and managing that is not a good idea: this prevents reallocation of memory to the uses for which it is in greatest demand.

We propose the use of independent pageout daemons for the management of various cluster memory pools. These daemons can respond appropriately to the potentially different memory demands from cluster to cluster. Any of several standard paging algorithms can be used to implement the daemons [18]. The migration and replication costs can be dynamically modified to feed information about page availability back into the replication and migration algorithms. These modifications should be restricted to increasing costs above their basic levels to reflect page shortages and hence discourage future use of memory in clusters with page shortages. Decreasing migration costs to encourage freeing memory in clusters with shortages, and cost-based reclamation of replicates are fraught with potential danger; this is because not all system components that use memory are or can be sensitive to costs – hence these cost-driven alternatives may result in heavily used pages being evicted in order to retain lightly used ones for cost-insensitive components.

5.2 Memory Management Interactions

Algorithms **M** and **R** can be incorporated into the operating system's memory management code on a NUMA multiprocessor. Implementing these algorithms inside the operating system allows their benefits to accrue to all uses of the machine, but also results in interactions with other memory management functions that must be dealt with as part of the implementation.

We use the virtual memory management portion of the Mach operating system [12] as a base for a case study of these interactions. Mach is a multiprocessor operating system developed at Carnegie-Mellon University; its VM system provides advanced memory management functionality including flexible sharing (both read/write and virtual copy), mapped files, and external memory management. This functionality stresses the interactions of our algorithms with the remainder of the operating system, and serves to expose potential problems.

The Mach VM implementation is cleanly split into machine-independent and machine-dependent portions. The machine-dependent portion consists of a single module, the *pmap* module, that is responsible for all physical map operations. The machine-independent portion of the system associates a pmap with each address space and invokes the pmap module as needed to perform mapping operations. Mach supports parallel execution of multiple threads within a single task's address space; this parallel execution can result in a single pmap being used simultaneously by more than one processor.

Mach envisions support for non-uniform physical memory by adding a NUMA layer between the machine-independent and machine-dependent portions of the VM system [18]. This layer hides the non-uniformity of the memory structure by translating *logical pages* (manipulated by the machine-independent portion of the VM system) to physical pages (in the hardware) in order to implement architecture-specific memory management policies (e.g. replication, migration). A similar translation process is needed for pmaps to allow replication within a single address space if its threads are spread across multiple clusters: in this case each cluster would have its own physical map, but the collection of these pmaps would appear as a single logical pmap to the machine-independent portion of the VM system. This adds additional complexity to the NUMA layer to better support multi-threaded applications, and may complicate interfaces that allow users to modify replication and migration behavior because an address space no longer uniquely specifies a cluster.

There are two other minor interactions of the NUMA layer with the remainder of the Mach VM system, and one major interaction. The two minor interactions are:

- Pageout functionality must be moved into the NUMA layer and redesigned to use multiple pageout daemons as discussed in Section 5.1. The resulting daemons must cope with system-wide (logical) page shortages as well as page shortages in the individual clusters.
- There must be a physical page available for every free logical page. Therefore use of a physical page for replication may require stealing a logical page from the resident page subsystem to maintain this invariant. Freeing of such a replicate should cause the stolen logical page to be returned to the resident page subsystem's free list.

Neither interaction poses great difficulties for an implementation.

The major interaction involves replication and copy-on-write. If the system has replicated a shared page that must be copied if written, then the replicates can be used to satisfy write faults on the page; this avoids the costs of creating an extra copy, but imposes extra costs if the replicate was used by more than one address space and has to be recreated as a result. The easy case is if there is a replicate that is only being used by the address space that caused the write fault; this replicate can always be used to satisfy the fault. For multiple address spaces, we would propose always using the replicate unless one of the other spaces has indicated that the replicate is needed (cf. the **always replicate** operation in section 10). An additional primitive must be added to Mach's machine-dependent interface to implement this functionality; the fault handler must be able to find out if the NUMA layer has a replicate that can be used to satisfy a write fault.

6 Hardware Support

Existing multiprocessor hardware will not allow a sufficiently accurate implementation of the NUMA memory management schemes discussed in this paper. Software systems that impose a level of indirection on all accesses to memory or shared memory can not hope to recover from this performance penalty. Thus we propose an architecture with hardware support for our algorithms. For each page, a set of two

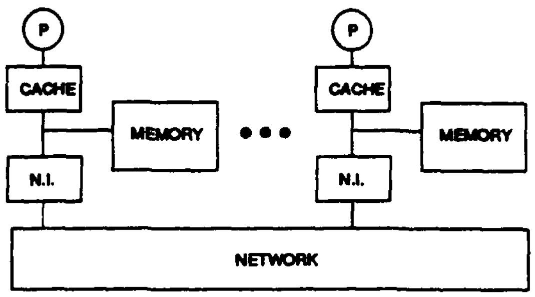

Figure 1: Architectural model

reference counters is required per processor in the system. Together with increment/decrement logic these maintain the counts required by algorithms **R** and **M**. An exception is caused when the built-in threshold for migration or replication is reached. The operating system then deals with the copying and remapping operations required.

The counters are kept with their associated memory page. For a 64-processor system and 16-bit counters, we thus require 256 bytes of memory per page. This translates to 50% overhead for 512-byte pages, 25% overhead for 1K pages, 6.25% for 4K pages and 3.125% for 8K pages. The overhead seems quite acceptable for pages in the 4K - 8K range.

For replication, we only need to increment a single counter. Migration, on the other hand, requires updating two counters, one of which must be chosen from the non-zero counters for that page (local references only update the latter counter). Since the updating of the counters must take place transparently and at the same speed of a memory reference, a sequential search for non-zero counters is not acceptable. An alternative is to pick a counter and decrement it if it is non-zero. This is much simpler to implement and our simulations indicate that its performance is similar to the original migration algorithm.

Copy on reference is the major alternative to our replication approach. It should be used where enough locality is known (e.g. from previous experimentation) or expected (e.g. code) to exist to cause replication by algorithm **R**; if replication is going to occur, it is always more efficient to do it in response to the first reference. The proposed algorithm **R**, however, has an advantage in cases where read-only data may not be accessed enough to cause replication: systems that manage large amounts of data for which locality cannot be assumed are an example. The choice of approach should depend on the situation being faced; copy on reference is probably more applicable to the most common situations than our delayed replication approach.

7 Performance Analysis

7.1 Architectural Model and Assumptions

The architectural model shown in Fig 1 was used for the analysis of the algorithms presented in this paper. It consists of several nodes linked by an interconnection network. Each node has a network interface(N.I.), its share of the global memory, a processor and a cache. In the case of the NUMA architecture, the cache is write-through and is only used to cache memory locations in the local portion of the global memory. Global cache consistency is thus assured. The following costs were used for the various operations in our simulations:

Operation	Time
local reference	0.1 μs
remote reference	4.0 μs
replication of a page	1200 μs
migration of a page	2100 μs

The access costs are based on those found in the Butterfly; replication and migration costs were estimated by examining page fault overheads in Mach (e.g. replication is very similar to a copy on write fault). These times include the overhead of updating page tables: this results in larger migration costs because more page tables must be changed by a migration than by a replication.

We also evaluate a system that allows the caches to cache *all* memory locations and uses a directory-based scheme to keep the caches coherent [1]. The hardware requirements of this scheme are greater, both in terms of memory requirements and in terms of complexity of the directory controller. We assume the cache scheme has the following costs for comparison with the NUMA scheme:[2]

Operation	Time
cache hit	0.1 μs
cache miss	4.0 μs
invalidation	4.0 μs

The algorithms were evaluated using multiprocessor traces of three parallel applications: LocusRoute [13], MP3D [11] and P-THOR [17]. LocusRoute is a standard cell global router, which exploits parallelism at a fairly coarse grain. MP3D is a 3-dimensional particle simulator. It uses distributed loops and is a typical example of parallel scientific code. P-THOR is a parallel logic simulator.

The traces were gathered on a VAX 8350, using a combined hardware/software scheme [7]. All traces were 8-processor runs and contain about half a million references per processor (4 million references total).

A simulator was used to keep track of the location of every memory page and the values of the various counters. The initial placement of each page was random. Code pages were allowed to replicate while data pages could only migrate.

7.2 Results

Figure 2 shows the performance increases gained by applying replication and migration. We are plotting the overall runtime for four schemes. "Neither" designates a random placement of memory pages in the nodes with neither replication nor migration allowed. The other points show the effect of allowing only migration, only replication and then both. Each curve shows the results for one of the three applications. When both replication and migration are allowed, the overall runtime decreases by a factor of 5 to 10.

We also explored variations of three parameters: page size, replication threshold and migration threshold. The results from varying the page size are shown in Figure 3. We plot

[2] Note that the cost given for invalidation is a *per remote invalidation* cost. Thus if a write reference results in invalidations in three remote caches, the total cost is assumed to be 12 μs.

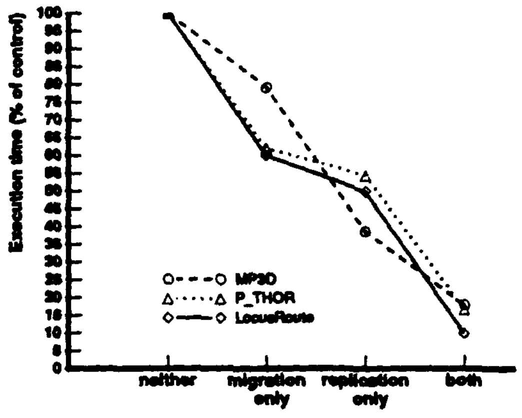

Figure 2: Performance Improvements

the effect of varying page size against the simulated run time of the trace. This time is shown as a percentage of the time required to execute the trace with replication and migration turned off (i.e. "neither" in Fig. 2).

Two effects are important when deciding the most efficient page size. Smaller pages are basically smaller units of replication/migration and would be expected to efficiently track the sharing needs of a program. At the same time, however, the fixed portion of remapping overhead makes larger pages more efficient. These two effects result in a U-shaped curve as seen in Figure 3. Although the position of the curves for the different applications varies vertically, their shape is basically identical. In every case the best page size was 512 bytes, but the effect of using larger pages was not significant.

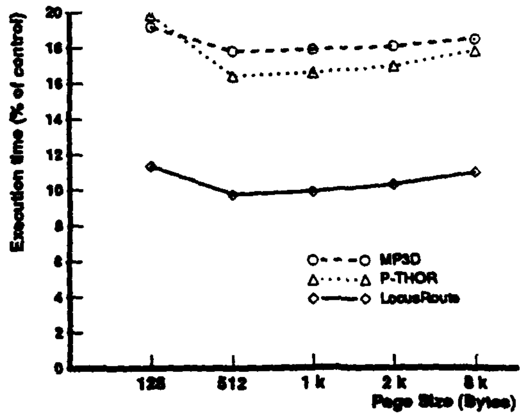

Figure 3: Effect of page size

Tuning the thresholds in these algorithms to match expected access patterns may improve average case performance without sacrificing constant factor bounds on the worst case performance. Tuning increases the constant factors in the bounds (i.e. the resulting algorithms are no longer competitive), but the increases may be offset by the improved average case behavior. For example changing the replication threshold in algorithm R from r to $0.5r$ or $2r$ increases the constant factor in the performance bound from 2 to 3. Our results show that threshold tuning has very little effect on overall performance. In each case lowering the threshold increases performance by a very small amount. Most of the pages are replicated or migrated just once, so the sooner the movement takes place, the lower the overall cost.

In the results presented above, each page was allowed to migrate any number of times. We also explored a variation where only a single migration per page was allowed – this basically allowed the program to achieve a good initial page assignment. The performance of this variation was just as good as when multiple migrations were allowed, indicating that a good initial assignment is the most critical factor. This may be due in part to the length of the traces. Longer traces may show a larger benefit for dynamic migration, as the program moves from one "working set" to another.

Tables 1 and 2 compare the performance of the NUMA memory management scheme to that of a directory-based cache scheme. Due to limitations of space, only results for LocusRoute are shown, but the relative performance was similar for the other two applications. The data shows that the cache scheme does about twice as well as the NUMA scheme. While cost for local references are comparable, the extra cost of remote references in the NUMA scheme is not offset by the extra cost of misses and invalidations in the cache scheme.

Table 1: NUMA scheme performance

NUMA		
Count	Operation	Cost (μs)
36	replication	43,200
86	migration	180,600
227,304	remote ref	909,216
4,114,180	local ref	411,418
	Total	1,544,434

Table 2: Cache scheme performance

CACHE		
Count	Operation	Cost (μs)
31,435	read miss	125,740
8,547	write miss	34,188
5,192	invalidation	20,768
4,301,493	hit	430,149
	Total	610,845

8 Extensions to Other Architectures

Competitive replication and migration algorithms have been found for certain extensions to our basic architectural model. A companion paper [2] presents competitive algorithms for replication and migration in arbitrary trees and architectures based on trees including hypercubes and meshes. The related topologies of rings and torii handle replication easily, but pose problems for migration.

Migration on rings and torii (products of rings) is problematic. Bidirectional rings exhibit the phenomenon of *pinning* [15] in which accesses in both directions from the far side of the ring can pin a page in place and prevent it from migrating closer to the accesses. Unidirectional rings or *unidirectional* routing structures imposed on bidirectional rings avoid this problem, but instead exhibit the phenomenon of *cycling* in which a static access pattern distributed over the ring can cause a page to cycle around the ring interminably (using up ring bandwidth) when it should stay put. It is possible that more sophisticated algorithms that keep additional information about the pattern and history of accesses can avoid these problems, but this extra state and the cost of updating it may affect the overall utility of such algorithms.

9 Replication of Writable Pages

So far we have not allowed the replication of writable pages. For portions of shared memory that are rarely written (called *mostly-read objects* in [20]), the amortized costs of the atomic updates required by the writes may not be prohibitive. Such a scheme can be implemented by using hardware mechanisms to cause a trap if a write occurs to any of the replicates. The handler for this trap can then perform the atomic update by disabling all access to all copies until the write has been propagated to all of them. Relaxed consistency constraints are preferable if the data has to be updated frequently. On the other hand, if the memory is never written after some point, then replication is a very good idea. Researchers working on the ACE project at IBM Hawthorne have found this to be the case for a parallel shortest path program: the data structures describing the graph to be searched are never written after the initialization phase, but are read heavily during the search. Replicating these structures into local memories on their machine produced major improvements in the run time of the application [3].

Algorithm R may not be appropriate for managing replicated writable shared memory because it ignores the costs of updating other replicates in response to a write. The *General-Snoopy-Caching* algorithm in [8] is a better choice if these costs are important because it takes them into account: this algorithm is competitive with a competitive factor of 3. If update costs depend on the number of replicates (e.g. if individual messages are required to update each replicate), then the algorithm must be modified accordingly in order to remain competitive.

10 Input and Feedback

If additional information is available about the access patterns for a page, the algorithms M and R can be further improved upon. We propose four primitives to help specify this additional memory usage information. The actual information may be provided by the user directly or it may come as feedback from a profiler. The primitives are:

Never replicate: On average, this page is used so infrequently in this cluster that it should never be replicated, even if it accumulates r accesses.

Always replicate: On average, this page will be used enough in this cluster to justify replication as early as possible. Alternatively, this page is read-only due to the use of copy-on-write techniques and is going to be written (which will require a copy to be made).

Never migrate: On average, this page is used so infrequently that it should not be migrated to this cluster even if it accumulates enough accesses to justify migration.

Anchor: This page will be so heavily used in this cluster that it should be anchored here and not allowed to migrate until further notice. An option to reverse this effect is also needed.

Lazy evaluation can be used to delay the effects of **always replicate** until the memory in question is actually accessed. This is done by unmapping the page in hardware and performing the operation in response to the page fault generated by the first access. This permits greater flexibility in the use of this primitive, as no additional cost is imposed for pages that are not used; similar functionality is provided by copy on reference.

These primitives can also be used to provide feedback from the management algorithms and other instrumentation over multiple runs of an application to improve its performance by adapting its memory usage to the memory structure of the machine. This feedback may reduce the effort required to restructure data to take advantage of non-uniform memory architectures.

11 Related Work

Competitive management of distributed shared memory is a topic at the juncture of several active areas of research. Li [9], Cheriton [4], and others have implemented distributed shared memory using messages on a network. The hardware for these implementations does not support remote accesses or access forwarding; this removes the choice of the amount of data to send in response to a request that is critical to our work. Most research projects in the area of NUMA architectures have implemented a shared memory programming model; the best known is BBN's Uniform System [19], and it typifies them in that it directly exports the non-uniform memory structure to users. Our work supports automatic management mechanisms that free users from some of the details involved in managing non-uniform memory, and should make these machines easier to program. Scheurich and Dubois [15] have independently discovered an extension of our migration algorithm to mesh-connected machines and hypercubes, but not its competitive properties. They also note the pinning problem for bidirectional rings, but not the cycling problem for unidirectional rings. Rudolph and Segall [14] are investigating a bus-based hardware consistency mechanism for pages. Their work differs from ours in that it depends on a hardware consistency mechanism to permit replication of writable pages without weakening consistency. Finally our work makes contributions to the area of competitive algorithms; the migration algorithms are competitive solutions to several cases of the 'one server with excursions' problem [10]. While we would like to solve this problem in full generality (i.e. for any topology), we are of the opinion that any such solution must maintain too much state to be applicable to real systems. Finally the techniques of competitive algorithm analysis may be applicable to other resource management problems that occur in distributed systems and multiprocessors, such as load balancing [6].

12 Conclusion

This paper has presented and analyzed algorithms for managing memory in NUMA multiprocessors and related systems. Competitive algorithm analysis guarantees small constant factor bounds on performance with respect to optimal algorithms that require information on future memory reference behavior. A case study of the Mach VM system indicates that incorporation of these algorithms into an operating system kernel should not pose any great difficulties. In contrast, hardware support is required to obtain the full functionality of our approach on most multiprocessors. We have also sketched extensions of our approach to additional hardware architectures (e.g. hypercubes) and software programming models (e.g. weak consistency).

We used trace driven simulations to evaluate our approach and compare it to other alternatives. Speedups of 5 to 10 over random assignment of pages are achieved on production applications without modifying the applications for NUMA architectures. These results indicate that significant instruction and data locality may be present in many shared memory multiprocessor applications, and that this locality can be exploited automatically. We also compare our proposed hardware support with the more aggressive approach of fully-consistent caches. An additional factor of 2 in performance can be obtained from the cache approach, but at the cost of much more hardware.

Acknowledgements

Most of the theoretical results in this paper represent joint work with Daniel Sleator; complete proofs and details can be found in [2]. We would also like to thank Richard Rashid and Roberto Bisiani for encouragement and support. Anoop Gupta and Wolf-Dietrich Weber are supported by DARPA contract N00014-87-K-0828. Anoop Gupta is also supported by a faculty award from Digital Equipment Corporation.

References

[1] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *15th International Symposium on Computer Architecture*, 1988.

[2] D. Black and D. Sleator. Algorithms for the 1-Server problem with Excursions. Technical report, Computer Science Dept., Carnegie Mellon University, Pittsburgh, PA, 1988. to appear.

[3] W. Bolosky. Personal Communication, September 1988.

[4] D. Cheriton. Unified Management of Memory and File Caching Using the V Virtual Memory System. Technical Report STAN-CS-88-1192, Computer Science Dept., Stanford University, Stanford, CA, 1988.

[5] W. Crowther, J. Goodhue, E. Starr, R. Thomas, W. Milliken, and T. Blackadar. Performance Mmeasurements on a 128-node Butterfly Parallel Processor. In *Intl. Conf. on Parallel Processing*, pages 531-540, 1985.

[6] A. Ezzat. Load Balancing in NEST: a Network of Workstations. In *Fall Joint Computer Conference (FJCC)*, November 1986.

[7] S. Goldschmidt. Simulating Multiprocessor Memory Traces. EE390 Report, Stanford University, Dec. 1987.

[8] A. Karlin, M. Manasse, L. Rudolph, and D. Sleator. Competitive Snoopy Caching. Technical Report CMU-CS-86-164. Computer Science Dept., Carnegie Mellon University, Pittsburgh, PA, 1986. Preliminary version appeared in 27th FOCS, 1986.

[9] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. In *5th Symp. on Principles of Distributed Computing*, pages 229–239, 1986.

[10] M. Manasse, L. McGeoch, and D. Sleator. Competitive Algorithms for Server Problems. In *20th Symp. on Theory of Computing*, pages 322–333. 1988.

[11] J. McDonald. A Direct Particle Simulation Method for Hypersonic Rarified Flow on a Shared Memory Multiprocessor. CS411 - Final Project Report, Stanford University, Mar. 1988.

[12] R. Rashid, A. Tevanian Jr., M. Young, D. Golub, R. Baron, D. Black, J. Chew, and W. Bolosky. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Archtectures. *IEEE Trans. Comput.*, 37(8):896–908, August 1988.

[13] J. Rose. LocusRoute: A Parallel Global Router for Standard Cells. In *Design Automation Conference*, pages 189–195, June 1988.

[14] L. Rudolph and Z. Segall. Dynamic Paging Schemes for MIMD Parallel Processors. Research notes on work in progress.

[15] C. Scheurich and M. Dubois. Dynamic Page Migration in Multiprocessors with Distributed Global Memory. In *Int. Conf. on Distributed Computer Systems*, pages 162–169, 1988.

[16] D. Sleator and R. Tarjan. Amortized Efficiency of List Update and Paging Rules. *Commun. ACM*, 28(2):202–208, February 1985.

[17] L. Soule and T. Blank. Parallel Logic Simulation on General Purpose Machines. In *Design Automation Conference*. pages 166–171, June 1988.

[18] A. Tevanian Jr. *Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, December 1987.

[19] R. Thomas and W. Crowther. The Uniform System: An approach to runtime support for large scale shared memory multiprocessors. In *Proc. of 1988 Int. Conf. on Parallel Processing, Vol II*, pages 245–254, 1988.

[20] W.-D. Weber and A. Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, Apr. 1989.

[21] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *11th Symp. on Operating Systems Principles*, pages 63–76, 1987.